Strutture dati elementari - Esercizi

Pietro Di Lena

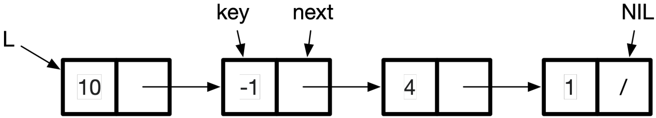
Dipartimento di Informatica – Scienza e Ingegneria
Università di Bologna

Algoritmi e Strutture di Dati
Anno Accademico 2021/2022

Esercizio 1

- Scrivere un algoritmo che, dati in input un intero positivo k ed una lista concatenata semplice, restituisca il k-esimo elemento a partire dalla fine

- Esempio, se $k = 1$ l'algoritmo deve restituire l'ultimo elemento; se $k = 2$ il penultimo, e cosi via



Esercizio 1 - Soluzione

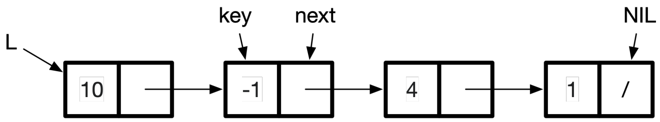
```
1:  function LASTK(List L, Int k) → List
2:      tmp = L                              ▷ Temporary pointer
3:      n = 0                                ▷ Number of nodes
4:      while tmp ≠ NIL do                   ▷ Count the number of nodes
5:          n = n + 1
6:          tmp = tmp.next
7:      if n < k then
8:          return NIL
9:      else
10:         tmp = L                          ▷ Reset tmp
11:         n = n − k + 1                    ▷ Index of last-k node
12:         i = 1
13:         while i < n do                   ▷ Search the last-k node
14:             tmp = tmp.next
15:             i = i + 1
16:         return tmp
```

- Costo computazionale: $\Theta(n)$ (visitiamo sempre tutta L: linee 4-6)
- $n =$ numero di nodi nella lista

Esercizio 2

- Scrivere un algoritmo che, dati in input un intero positivo k ed una lista concatenata semplice restituisca il k-esimo elemento a partire dalla fine senza visitare per due volte la lista (per memorizzarne la lunghezza)

- Esempio, se $k = 1$ l'algoritmo deve restituire l'ultimo elemento; se $k = 2$ il penultimo, e cosi via



Esercizio 2 - Soluzione

```
function LASTK(List L, Int k) → List
    tmp1 = L                                  ▷ Temporary pointer
    tmp2 = L                                  ▷ Temporary pointer
    i = 0                                     ▷ Counter
    while tmp1 ≠ NIL and i < k do             ▷ Move tmp2 on the k-th node
        i = i + 1
        tmp1 = tmp1.next
    if i < k then                             ▷ There are less than k nodes
        return NIL
    else
                        ▷ The distance between tmp1 and tmp2 is exactly k
        while tmp1 ≠ NIL do
            tmp1 = tmp1.next
            tmp2 = tmp2.next
        return tmp2
```

- Costo computazionale: $\Theta(n)$ ($tmp1$ visita sempre tutti i nodi di L)
- $n =$ numero di nodi nella lista

Esercizio 3

- Scrivere un algoritmo ricorsivo che, data una lista concatenata semplice, la modifichi eliminando ogni elemento pari
- Esempio, se $L = [4, 6, 7, 3, 2, 5]$ al termine dell'esecuzione $L = [7, 3, 5]$

Esercizio 3 - Soluzione

```
function deleteEven(List L) → List
    if L == NIL then
        return NIL
    else if L.key mod 2 == 0 then                    ▷ Even node, remove
        return deleteEven(L.next)
    else                                             ▷ Odd node, no remove
        return L.next = deleteEven(L.next)
        return L
```

- Costo computazionale: $\Theta(n)$ (visita sempre tutti i nodi della lista)

$$T(n) = \begin{cases} 1 & n = 0 \\ T(n-1) + 1 & n > 0 \end{cases}$$

- n = numero di nodi nella lista

Esercizio 4

- Scrivere un algoritmo ricorsivo che, data una lista concatenata semplice, la modifichi eliminando ogni elemento pari e replicando ogni elemento dispari tante volte quanti sono gli elementi pari che lo precedono

- Esempio, se $L = [4, 6, 7, 3, 2, 5]$ al termine dell'esecuzione abbiamo che $L = [7, 7, 7, 3, 3, 3, 5, 5, 5, 5]$

Esercizio 4 - Soluzione 1

```
function deleteAndDuplicate(List L, int nPrec) → List
    if L == NIL then
        return NIL
    else if L.key mod 2 == 0 then                          ▷ Even node, remove
        return deleteAndDuplicate(L.next, nPrec + 1)
    else                                                   ▷ Odd node, duplicate
        L.next = deleteAndDuplicate(L.next, nPrec)
        while nPrec > 0 do                                 ▷ Duplication with nPrec head insert
            tmp = new List(L.key)
            tmp.next = L
            L = tmp
            nPrec = nPrec − 1
        return L
```

- Soluzione mista ricorsiva/iterativa
- Prima esecuzione: deleteAndDuplicate$(L, 0)$

Esercizio 4 - Soluzione 2

```
function deleteAndDuplicate(List L, int nPrec, int nDup) → List
    if L == NIL then
        return NIL
    else if L.key mod 2 == 0 then                                  ▷ Even node, remove
        return deleteAndDuplicate(L.next, nPrec + 1, nPrec + 1)
    else if nPrec > 0 then                                         ▷ Odd node, duplicate
        tmp = new List(L.key)
        tmp.next = deleteAndDuplicate(L, nPrec, nPrec − 1)
        return tmp
    else                                                           ▷ Odd node, no duplicate
        L.next = deleteAndDuplicate(L.next, nPrec, nPrec)
        return L
```

- Soluzione interamente ricorsiva
- Prima esecuzione: deleteAndDuplicate($L, 0, 0$)

Esercizio 4 - Analisi del costo computazionale

- Vale sia per la versione mista che per la versione solo ricorsiva
- Assumiamo $n =$ numero di nodi nella lista
- Costo ottimo: $\Theta(n)$ (ci sono solo nodi pari o solo nodi dispari)
- Caso pessimo (irrealistico): ogni nodo viene duplicato tante volte quanti sono i nodi che lo precedono

  $$T(n) \leq \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} = \frac{n^2 - n}{2} = O(n^2)$$

  Quindi il costo pessimo T(n) è limitato superiormente da $O(n^2)$
- Caso effettivo: un nodo pari ed un nodo dispari, alternati

  $$T'(n) = \sum_{i=1}^{n/2} i = \frac{n/2(n/2+1)}{2} = \frac{n^2}{8} + \frac{n}{4} = \Theta(n^2)$$
- Poiché $T'(n) = O(T(n))$ concludiamo che il costo pessimo è $\Theta(n^2)$

Esercizio 5

- Scrivere un algoritmo che, preso in input un albero binario, ne cancelli tutte le foglie

- L'algoritmo ritorna il puntatore alla radice dell'albero (che potrebbe essere NIL se l'albero consiste di una sola foglia)

Esercizio 5 - Soluzione

```
function REMOVELEAVES(TREE T) → TREE
    if T == NIL or ISLEAF(T) then
        DELETE(T)
        return NIL
    else
        T.left = REMOVELEAVES(T.left)
        T.right = REMOVELEAVES(T.right)
        return T
```

- Costo computazionale: $\Theta(n)$ (visita tutti i nodi dell'albero)
- n = numero di nodi nell'albero

Esercizio 6

- Scrivere un algoritmo che, preso in input un albero binario, ritorni la somma dei valori contenuti nelle foglie

- Se l'albero è vuoto, l'algoritmo ritorna 0

- Risolvere l'esercizio sia con un algoritmo ricorsivo che iterativo

Esercizio 6 - Soluzione ricorsiva

```
function SUMLEAVES(TREE T) → INT
    if T == NIL then
        return 0
    else if ISLEAF(T) then
        return T.val
    else
        return SUMLEAVES(T.left)+SUMLEAVES(T.right)
```

- Costo computazionale: $\Theta(n)$ (visita tutti i nodi dell'albero)
- n = numero di nodi nell'albero

Esercizio 6 - Soluzione iterativa

```
function COUNTLEAVES(TREE T) → INT
    n = 0
    LET Q BE A QUEUE
    if T ≠ NIL then
        ENQUEUE(Q, T)
    while Q.size ≠ 0 do
        x = DEQUEUE(Q)
        if ISLEAF(x) then
            n = n + x.val
        if x.left ≠ NIL then
            ENQUEUE(Q, x.left)
        if x.right ≠ NIL then
            ENQUEUE(Q, x.right)
    return n
```

- Costo computazionale: $\Theta(n)$ (visita tutti i nodi dell'albero)
- n = numero di nodi nell'albero

Esercizio 7

- Scrivere un algoritmo che, preso in input un albero binario, elimini tutte le foglie che sono figli sinistri e con contengono lo stesso valore del nodo padre

- L'algoritmo non ritorna nessun valore (la radice non viene mai modificata dato che non ha un padre)

Esercizio 7 - Soluzione

```
function DELLEAVES(TREE T)
    if T ≠ NIL then
        if T.left ≠ NIL and ISLEAF(T.left) and T.left.val == T.val then
            DELETE(T.left)
            T.left =NIL
        DELLEAVES(T.left)
        DELLEAVES(T.right)
```

- Costo computazionale: $\Theta(n)$ (visita tutti i nodi dell'albero)
- n = numero di nodi nell'albero

Esercizio 8

- Scrivere un algoritmo che calcoli l'altezza di un albero binario
  - Se l'albero contiene solo la radice, l'altezza è 0
  - Se l'albero è vuoto, l'altezza è -1
- Risolvere l'esercizio sia con visita in profondità che in ampiezza

Esercizio 8 - Soluzione in profondità

```
function height(Tree T) → int
    if T == NIL then
        return −1
    else if isLeaf(T) then
        return 0
    else
        return 1+max(height(T.left),height(T.right))
```

- Costo computazionale: $\Theta(n)$ (visita tutti i nodi dell'albero)
- n = numero di nodi nell'albero

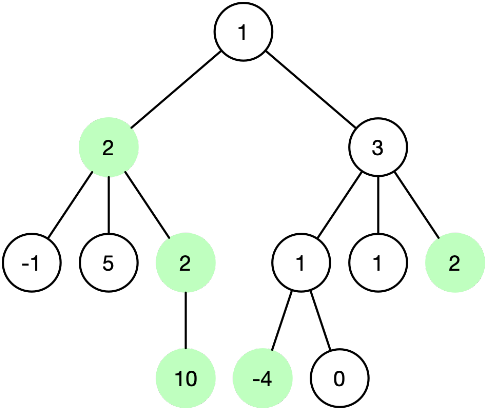
Esercizio 8 - Soluzione in ampiezza

```
function HEIGHT(TREE T) → INT
    n = −1
    if T ≠ NIL then
        LET Q BE A NEW QUEUE
        ENQUEUE(Q, [T, 0])
        while Q.size ≠ 0 do
            [x, n] = DEQUEUE(Q)
            if x.left ≠ NIL then
                ENQUEUE(Q, [x.left, n + 1])
            if x.right ≠ NIL then
                ENQUEUE(Q, [x.right, n + 1])
    return n                    ▷ Last visited node is the deepest one
```

- Costo computazionale: $\Theta(n)$ (visita tutti i nodi dell'albero)
- n = numero di nodi nell'albero

Esercizio 9

- Scrivere un algoritmo che conti il numero di nodi con valore pari in un albero generico (non binario)



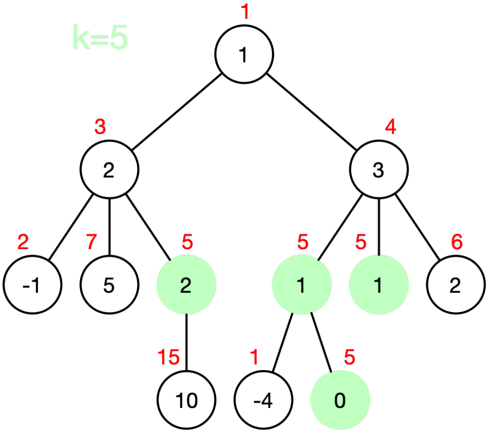
Esercizio 9 - Soluzione

```
function COUNTEVEN(TREE T) → INT
    if T == NIL then
        return 0
    else if T.val mod 2 == 0 then
        return 1+COUNTEVEN(T.first)+COUNTEVEN(T.next)
    else
        return COUNTEVEN(T.first)+COUNTEVEN(T.next)
```

- Costo computazionale: $\Theta(n)$ (visita tutti i nodi dell'albero)
- n = numero di nodi nell'albero

Esercizio 10

- Scrivere un algoritmo che, dato un albero generico (non binario) ed un intero l, conti il numero di nodi tali per cui la somma dei valori del percorso radice-nodo sia uguale a k



Esercizio 10 - Soluzione

```
function COUNTK(TREE T, INT k) → INT
    if T == NIL then
        return 0
    else
        if T.val == k then
            return 1+COUNTK(T.first, k − T.val)+COUNTK(T.next, k)
        else
            return COUNTK(T.first, k − T.val)+COUNTK(T.next, k)
```

- Costo computazionale: $\Theta(n)$ (visita tutti i nodi dell'albero)
- n = numero di nodi nell'albero